Enabling Web Services

Phillip J. Windley, Ph.D.
Chief Information Officer
Office of the Governor
State of Utah

Web services are a topic that has garnered a lot of attention in the last few years. The term has numerous meanings depending on who you ask. To some, there's not much "web" in "web services." Overall, however, the term is used to describe the ability to easily link programs and data from various sources in a way that creates a new look at the data or even a new application.

One hope is that web services will enable the kinds of cross-organizational applications that have been our focus over much of the past year. While individual components and their participation as a web service will be decided on a case by case basis, there are some things we can do with relatively little effort that will enable the easy sharing of data between agencies, across levels of government, and even with private industry. What's more, this same data can be incorporated into multiple applications with little additional effort.

The most important thing we can do is to realize that we are the keepers of large amounts of data and to ensure that any time we make that data available on the network (whether publicly or not) we should do so in a way that gives that data a unique name and preserves the structure of the data in a way that's easy to use.

This paper presents a list of principles for enabling web services that, if followed, will ensure greater flexibility in data resources. This flexibility will allow data to be used in ways that we cannot envision now at *little* additional cost. This paper also discusses technologies and techniques that make these principles work. Please remember that this paper is merely an introduction to these technologies. The resources box gives further information for those who wish to go further.

Principles for Enabling Web Services

Enabling web services requires that we make data available for use by applications without knowing *a priori* exactly how that data will be used. How can we do that? By being smart about how we design our new data sources and about how we enable access to our legacy data sources. The following list of principles will help ensure our data sources will be useful in a wide variety of circumstances, not just those for which they were initially designed. For the most part, adhering to these principles should not appreciably increase the cost.

1. *Every* data record and collection is a resource.
2. *Every* resource should have a URI.
3. Cool URI's *don't* change.
4. Data queries on existing resources should be done with a GET.
5. Use POST to create new resources.

6. Preserve the structure of data until the *last possible moment* (*i.e.* return XML).
7. Make DTD's available *online* for your XML.
8. Make data available in multiple flavors.
9. Use Metadata (RDF) for XML.
10. Document your service API using WSDL, WRDL, or some other standard.
11. Advertise the presence of the data using WSIL.
12. Adhere to data standards such as RSS where available.
13. Use HTTP authentication as much as possible.

The remainder of this paper will discuss these principles in detail and describe the technologies behind them.

Every Data Record and Collection is a Resource

When you use the Internet and click on a link, what you get back is formally called a *resource*. People often refer to resources as a "web pages' or "documents" because most often, what comes back is designed for human readability, but that's not always the case, so resource serves as a more general term.

By considering every data record and collection a resource, we are changing our mindset so that we view our data as something that is consumed by various programs (including, but not limited to, web browsers), often without our direct knowledge that the data is being used.

Every Resource Should Have a URI

Someone once remarked that every UPS package has its own homepage on the Internet. Everyone knows that you can go to www.ups.com and use the tracking number of a package to find its status, but have you ever thought of that status page as being the homepage for that package? In fact, the status page has a uniform resource indicator (URI) that identifies its unique location on the web and that URI can be linked in another document or bookmarked for later reference—just like any other web page. The package "homepage" is no different that any other homepage on the Internet in that regard.

The URI is the "web page address" that you type in the address box on your browser.[1] Every URI is unique and represents a "resource" on the web. URIs are one of the most important features of the web. Without URIs much of what we take for granted on the web wouldn't work. As a simple example, having a universal namespace created using URIs allows any document, anywhere on the web, to refer to any other document, anywhere on the web, without the authors of the two documents to having to agree on the same software package, or system

[1] Uniform Resource Indicator (URI) is a more general term for what has commonly referred to as a Uniform Resource Locator, or URL.

beyond what's inherent in the web itself. In fact, Paul Prescod has said: "If there is one thing that distinguishes the Web as a hypertext system from the systems that preceded it, it is the Web's adoption of a single, global, unified namespace."

Giving other resources, such as data records, a URI makes them part of this same universal namespace and ensures that they can take advantage of all the utility of the web as well.

URIs have three major components:

1. A protocol identifier followed by a colon (e.g. `http:`).
2. A domain name indicating a unique computing domain on the Internet (e.g. `www.dopl.utah.gov`)
3. A path component indicating what specific resource in that domain is to be identified (e.g. `/llp?ln=windley&lang=en`[2]).

There can be other components as well, including authentication information, port numbers, etc. but these three are the most common.

Resources:

The following links will take you to resources that will help in exploring the topics in this paper further.

URIs:

Cool URLs Don't Change.

Axioms of Web Architecture.

HTTP:

HTTP 1.1 Specification

Apache Web Server

XML:

XML.ORG Resources

XML and DTDs

W3C Document Object Model Page

XML.ORG DOM Resources

Apache XML Project

REST:

Second Generation Web Services

REST and the Real World

Fielding Dissertation

Metadata Standards:

Web Services Description Language

Web Resources Description Language

Web Services Inspection Language

WSIL and UDDI

Cool URI's *Don't* Change

The URI is the public interface for your resource and, consequently, deserves great thought. One of the key factors you should keep in mind when designing the URI for your system is that it should not change—ever. We cannot possibly know all the places that are linking to the resource and, consequently, cannot let them know when its name (the URI) changes, so the URI should be chosen so that it is meaningful and unlikely to change. As the system is updated and maintained, the non-volatility of the URIs should be preserved. Numerous tools and techniques exist to make this possible.

Designing the URIs for your information system should be one of the most important tasks of the design phase. It may seem unusual to think of designing URIs. After all, don't we just let the network folks tell us our domain name and let the path fall out however it may? Not in a well designed system. The last section talked about the three components that are typically part of a URL. All three are usually under our control and should be carefully chosen.

We should not construe this principle to mean that all resources need to be permanent. Just because URI's don't change, doesn't mean that the resource

[2] I use professional license queries as a running example in this paper because it represents a relatively pure data application on the web.

has to be always available. There are some resources that are transitory and some that go out of existence.

Data Queries on Existing Resources Should be Done with a GET

We've already stated that every resource should have an associated URI. For example, I should be able to query for a professional license using a URI like: `http://www.dopl.utah.gov/llp?ln=windley` (note: this is not a valid URI.) If this query returns a list of results, each of those results should be available individually using a URI reference. This implies that the data can be queried and retrieved using a Hypertext Transfer Protocol method called a GET.

Hypertext Transport Protocol or HTTP is the protocol of the web. HTTP is a URI aware protocol and, as such, cannot be used without URIs. HTTP is called an application protocol because it is designed for a specific application (hypertext transport, in this case) as opposed to the more general protocols of the Internet like TCP or UDP.

A protocol describes in very clear terms the interactions that two computers use when they talk to each other. In the case of HTTP, the kinds of requests and allowable responses are clearly identified. Other information, like control information, status codes, and error conditions, are also clearly described.

When a client (which may be a browser, mailer, or other web aware program) makes a request of a server, the server looks at the request type (method), performs the required action, and returns a result of the correct format. One of the great strengths of HTTP is that the requests are universal and response formats are very general.   In fact, HTTP can return any payload, including binary data, as its response.

For purposes of this document, I'm particularly interested in the kinds of requests that can be made with HTTP (for a full look at HTTP, see the specification in the "Resources" box). There are four primary request messages in HTTP 1.1:

1. GET – retrieve a resource (also can be thought of as a query).
2. POST – create a new resource (also can be thought of as changing the state of the server).
3. PUT – update a resource.
4. DELETE – delete a resource.

These four messages correspond to the kinds of messages that can be found in any database or information system. Because they correspond to the universal methods for interacting with data resources, HTTP can be used in very general ways to interact with legacy data systems.

Knowledgeable readers will no doubt be shaking their heads right now and saying "but browsers only use PUT and DELETE in very specific ways; how can we do general database work using a browser?" The point of this paper is to move our thinking beyond humans and browsers to web services. In a web services scenario, programs besides browsers will be interacting resources. Our goal is, wherever possible, to allow machine and human interaction concomitantly.

One point that deserves special mention is that the GET method assumes transparency, although this is not enforced in any special way in the protocol or in the servers that implement it. By transparent, we mean that the GET method should not change the state of the server in ways that are visible to the client. One reason it is important to respect this transparency requirement, as far as possible, is that proxy servers and gateways rely on it for proper operation. Because you cannot control what HTTP proxies may lie between you and the client, you have to assume that any GET request will be cached unless you take specific steps to ensure it will not[3].

By using a GET for queries and retrieval, we can take advantage of transparency to increase the scalability of our services through the use of caching. In addition, we increase the types of programs that can be used to retrieve the data or make the query. For example, if a query is expressed as a URI, it can be typed in a browser, linked in a HTML document, referenced in an XML document, queried from a program, and so on.

Use POST to Create New Resources

The POST method has a number of uses (and misuses). POST calls a resource and passes data to it that can be unmarshalled and used by the resource. The most frequent use of POST in browsers is to call an application on the web server to process data from a form.

One way to think of POST is that it is analogous to a CREATE (or INSERT in SQL) method in a database. There are actually more flexible uses than this, but using POST for creating new resources is a good start.

As an example of its misuse, POST can be used for data queries, but that would violate the last principle that requires that we use a GET for queries. Using POST for this would not be cachable (and hence increase the load on the primary server) and make it impossible to link to the query in a document or other resource.

REST

In the early days of the Internet, resources were difficult to find and when you did find them, you'd have to figure out what protocols and tools were required to access the resource before you could get to it. The web changed all of that; in what seemed like overnight, there was a universal way to address resources (the URI) and a universal protocol for retrieving them (HTTP). HTML even gave us a standard way of exchanging resources that pointed to other resources. This combination was very powerful and literally changed the world.

---

[3] Even if you do take these steps, it is likely that a poorly designed cache somewhere along the line (or even the user's browser) will cache it anyway. Best to avoid it if you can.

The way the web works, that is, its architecture, has a name: REST, which stands for Representational State Transfer. Roy Fielding, in his PhD dissertation (see resources box) defined the REST architecture and described REST as having the following four features:

- scalability of component interactions,
- generality of interfaces,
- independent deployment of components,
- and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

The web, as we know it, is not the only possible example of a RESTian architecture, but it is by far the largest and most well established example.

The four features that are outlined above come about because of very specific decisions that we made in designing the web. As designers of web applications, we must ensure that our decisions comply with RESTian principals to the largest extent possible or else we lose these advantages. To the extent we violate these principles, we make our application less useful, less scalable, and perform more poorly.

As an example of how we might design an application that violates the web architecture and thus is less useful than it might be, consider a professional licensing application for the State of Utah. One way to design the application, that follows the REST architecture, would be to ensure that each license has a URI (since it's a resource, give it a uniform *resource* identifier). As a result, a HTTP GET can retrieve the license. Another way would have a URI for a program than takes license ID data via a POST and then returns the license data.

While these two choices both use the technology of the web (HTTP), they do it in different ways and the decision changes how other programs can interact with the data. Consider three:

1. As a very simple example, consider that in the first instance, a general purpose search engine, such as Google, could compile an index of the licenses, whereas in the second, a special purpose search engine would have to be built.
2. Another example of how this choice makes a difference, consider that in the first instance, proxy servers are able to cache the returned data (so a program making heavy use of the data would see local copies) whereas in the second instance, the primary server has to handle the entire load.
3. A third example of how this choice makes the web less useful is to consider that in the first example, the URI is a universal name for the license, whereas in the second example the license data is only available to someone who knows about the proprietary namespace (license ID data). Special programming has to be done to interact with the private namespace.

The principals outlined in this paper are designed to help us make architectural and design choices as we design web applications so that we take full advantage of the REST architecture of the web.

Preserve the Structure of Data Until the *Last Possible Moment*

In a traditional web application, when a query is made and data is returned, the application on the server renders the result as HTML and returns it to the user. This principle asks that we delay changing the result into HTML until we absolutely have to. Another way of thinking of this principle is that all queries for data from a web server should produce at least XML to preserve structure. If human readability is required, post-process the XML to produce the HTML. As an example, if I go to the professional licensing division and query about doctors, the application should, at a minimum, produce XML.

For many, HTML seems the *lingua franca* of the web. HTML, or Hypertext Mark-up Language, is a set of tags that can be embedded in text to indicate to a browser or other program how to render the text on the screen (that's why its called a "mark-up language," it marks up the content with other information). One of the surprising things about the web is that HTML is not required. Unlike URIs, HTML is not part of the HTTP specification and HTTP works regardless of the payload. Thus, we can replace HTML with anything else that is useful.

Not long after the web began to get popular, people realized that there were times when it was useful for a program, rather than a human, to read the contents of a resource. The problem in this scenario is that HTML is built for describing how the content should be shown to humans and not very good at showing the underlying structure of the data being returned. Take for example, the following HTML snippet showing my address:

```
<B><A HREF="http://www.windley.com/"Phillip J. Windley</A></B><BR>
Office of the Governor<BR>
<FONT SIZE="-1">210 State Capitol</FONT><BR>
<FONT SIZE="-1">Salt Lake City, UT &nbsp;84042</FONT>
```

If I asked you to pick out my name or my ZIP code, you wouldn't have any problem because humans are remarkably adept at enforcing meaning on random strings of characters. A program would have a hard time making sense of this (except to render it for display). Even if we told the program that the ZIP code was the five digits in between the `&nbsp;` and the `</FONT>`, the result would be very brittle because I may decide to redo the display information contained in the HTML and break the program.

If, however, we formatted the information in a way that showed its structure, then a program would have no trouble picking out any piece of the address that we wanted:

```
<ADDRESS>
  <NAME>
   <FIRST>Phillip</FIRST><MI>J</MI><LAST>Windley</LAST>
   <LINK>http://www.windley.com</LINK>
  </NAME>
  <OFFICE>Office of the Governor</OFFICE>
  <STREET>210 State Capitol</STREET>
  <CITY>Salt Lake City</CITY>
  <STATE>UT</STATE>
  <ZIP>84042</ZIP>
</ADDRESS>
```

Notice that in the above text, it is easy for a program to grab the part between the `<ZIP>` tags. We've used the tags to indicate the structure of the data and we're unlikely to change the format as frequently as we would be if were marking up for display. In fact, we should take great care to design the schema so that it will not have to be changed and be reluctant to change it for trivial reasons.

This is the idea behind XML or eXtensible Mark-up Language. Notice that XML looks like HTML in that it has tags in angle brackets interspersed in the text. The purpose behind XML, however, is to display information *structure* rather than indicate proper display rendering. The mark-up language shown above is not XML, but rather a mark-up language for addresses that I made up using XML. XML is not a so much a mark-up language itself, but rather a framework for describing mark-up languages.

Make DTD's Available *Online* for Your XML

In XML, you describe your particular mark-up language using a document (itself expressed in XML) called a DTD or document type definition. The DTD allows an XML parser to parse an XML string and determine whether or not it is well formed. We should endeavor to document whatever XML format we output using a DTD and ensure that the up to date DTD is available online and referenced in the generated XML.

A well-formed XML string conforms to all of rules contained in the DTD. For example, in the address example I give above, you could specify that the `<FIRST>`, `<MI>`, and `<LAST>` tags all have to be children of the `<NAME>` tag and even things like `<LAST>` is required but the other two are optional. The technical details of writing DTDs are beyond the scope of this article, but the resources box has some excellent references.

XML is very regular and the standards surrounding it are well developed. Consequently, there are a number of XML parsers available to read XML. The parsers can, of course, tell you whether or not the XML is well formed, but, more importantly, they also create a parse tree that can be given to a program for using and manipulating the data in the XML. There is a standard format for the output of an XML parser called the Document Object Model, or DOM. Again, the details of the DOM are beyond the scope of this paper, but excellent references can be found in the resources box.

Make Data Available in Multiple Flavors

HTML gives us display and XML gives us data structure. From the previous discussion it might seem that you have choose between one or the other. The good news is that with XML, you can have both. Because XML is structured, data, it is relatively easy to write programs that translate it into any other format you desire. You could write your own program, but several standards are available. The most popular is called XSL (XML Stylesheet Language).

XSL is actually a programming language, but one aimed at the translation of XML from one format to another. XSL is rule based, meaning that it doesn't have the usual procedural statements of common programming languages. Instead the program is a series of rule definitions that associate a pattern and an action. When one of the patterns matches the piece of XML being translated at the moment, its associated action is applied to that XML.

Without going into the details of XSL, the following XSL program translates the address XML shown above into the equivalent HTML shown earlier:

```
<?xml version="1.0"?>
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml" indent="yes" encoding="us-ascii"/>

<xsl:template match="address">
<xsl:apply-templates select="name"/>
<xsl:value-of select="office"/><BR>
<FONT SIZE="-1"><xsl:value-of select="street"/></FONT><BR>
<FONT SIZE="-1"><xsl:value-of select="city"/>, <xsl:value-of
select="state"/>&nbsp;<xsl:value-of select="zip"/></FONT><BR>
</xsl:template>

<xsl:template match="name">
<B>
<A HREF=\"<xsl:value-of select="link"/>\">
<xsl:value-of select="first"/>
<xsl:value-of select="mi"/>
<xsl:value-of select="last"/>
</A></B><BR>
</xsl:template>

</xsl:stylesheet>
```

You can verify this yourself by copying the above lines into a file called address.xsl and then adding the following line to the beginning of the XML shown above and loading that file in your browser (most modern browsers support XSL translation of XML files natively):

```
<?xml-stylesheet href=”address.xsl" type="text/xsl"?>
```

Using a browser to do the translation isn't always practical (in fact, it is downright dangerous if you want reliability) and there are standalone XSL translators (XSLT) as well that can be made part of an application. One such program is called XALAN. The State has recently begun to use Interwoven for content management. Interwoven supports XSL translation natively.

Once we have the basic infrastructure in place to provide data as XML to preserve structure and translate it into HTML for rendering, we can consider making our data available in a number of different flavors. Using an XML translator, data can be returned in any number of formats by simply supplying a different translation. So, supporting multiple browser types and even machines with special format requirements is relatively easy. For example, a different XSL stylesheet could render the address XML shown above for display on a phone or with special accessibility features. We might also offer multiple XML renderings if there are multiple XML standards for the data format.

Use Metadata (RDF) for XML

A resource can contain almost anything. One of the most intriguing things a resource can contain is information about itself that helps other programs know what to do with it. Data that describes other data is called *metadata*. Metadata can include where the resource came from, what its format is, keywords, authors, and so on. Almost anything that someone might want to know about a resource can be stored in the metadata. On the web, metadata usually contains information about the resource that is machine readable. Machine readable means that the data is structured so that a computer can parse the data and know what to do with it.

As an example of metadata that is used by other programs, consider a search engine like Google. The way Google knows about documents is by scouring the web and retrieving every resource they can, reading it, and storing the location of the resource in a way that's associated with its content. Google and other search engines make use of metadata inside web resources to accurately perform this task.

The CIO's office and the State GILS project have endorsed the Dublin Core standards for RDF metadata. The GILS project provides RDF schema specific to the State of Utah that support the Dublin Core and is an excellent source of information and training about metadata. The resources box contains more information.

Document Your Service API Using WSDL, WRDL, or Some Other Standard

The idea behind offering XML as the primary format for data available on the web is that programs, in addition to people, will be able to access and use the data. For that to be possible, the interface to the data, that is what commands and conventions are used to access it, must be made clear to potential users. This interface is also called the API (application program interface). It makes sense to use XML to do this as well since at the very least, using XML ensures that the documentation will be parsable by the user and that it can be checked to ensure that it is well formed. Another way of thinking about this API documentation is as another kind of metadata for the resource.

At the time of this writing, I'm not convinced that there is a standard that fully meets our needs for documenting the data APIs that we will develop. There are two standards that might be useful: Web Services Description Language (WSDL) and Web Resources Description Language (WRDL). The problem with WSDL is that it is more oriented to remote procedure call (RPC) semantics like the SOAP protocol. While there may be SOAP based web services developed for state data resources, this paper is describing the foundational elements for such services. WRDL is closer to what we need to use to document data APIs. The problem with WRDL is that it is still very much in its early development and subject to change. Even so, since XSL will allow us to translate between formats (provided the content is similar) WRDL is not a bad choice given the philosophy that its better to document something than nothing and better to document in a structured manner than an *ad hoc* manner.

WRDL allows you to specify a resource type and give it a name (remember any data element could be a resource). Since this is a resource *type*, it represents a broad class of resources. Each resource type declares methods that act on it. WRDL allows you to document the methods (resources) in your API, give type information to the parameters, describe the output and describe its type.

The following gives an example of a WRDL file for a hypothetical professional licensing application named `llp` that accepts only GET methods with arguments named "`lid`" and "`ln`."

```
<?xml version="1.0"?>
<!DOCTYPE types SYSTEM "wrdl.dtd">
<types xmlns:xb="http://www.constantrevolution.com/xbind">
  <resourceType name="llp">
        <GET>
              <input>

                <query apiName="license_id" name="lid" type="integer"
default="prof_license"/>

                <query apiName="search_by_ln" name="ln" type="string">
                  <documentation>
                   search for license by last name.
                  </documentation>
                </query>

              </input>

              <output representations="prof_license">
              </output>
        </GET>
  </resourceType>

  <representationType name="prof_license" mediaType="text/xml">
        <xb:binding href="../xbind/prof_license.xbl"/>
  </representationType>

  <representationType name="html" mediaType="text/html">
  </representationType>

</types>
```

Notice that the WRDL documents the names of the arguments made to the get method, as well as the type of the argument and the return type. In this case, the only return type specified is "`prof_license`" which is defined in another document.

Advertise the Presence of the Data using WSIL

Once we have a service on the web and we've documented it so that others can use it, we still must ensure others can find it. Again, there are several emerging standards for doing this.

The one that gets the most press is called UDDI or Universal Discovery, Description, and Integration. UDDI specifies an active system of directory servers (like DNS servers, for example) that allow services to be registered, searched, and found. UDDI has had some trouble getting started and some have questioned its utility in a world where significant uses of web services will require contracts, service level agreements, and other contact between the provider and consumer of the service.

While that's being worked out, however, there is an alternative metadata specification for discovery of web services called Web Services Inspection Language, or WSIL. WSIL is useful in its own right and serves as a stepping stone to UDDI if it becomes widely available.

WSIL describes how a service requestor can discover an XML web service description on a web server, enabling such requestors to easily browse Web servers for XML web services. WSIL differs from WRDL because it only advertises the availability of the service; it doesn't describe how to use the service as WRDL does.

The great thing about WSIL is that it is built on the same technology that makes the web work. WSIL documents are XML resources, accessible via a URI. For any given organization, the root WSIL document has a standard name and lives in the root directory of the web server, so for utah.gov, our root WSIL document will live at

http://www.utah.gov/inspection.wsil

A WSIL document can link to resource descriptions (like WRDL resources) or to other WSIL documents. So, for example, the Division of Professional Licensing would document their services at

http://www.dopl.utah.gov/inspection.wsil

and the root document at www.utah.gov would link to it.

Here's an example of a WSIL document that links to a WRDL document and another WSIL document:

```
<?xml version="1.0"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
  <service>
    <description referencedNamespace="http://schemas.utah.gov/wrdl/"
                 location="http://wrdl.utah.gov/utah-weather.wrdl" />
  </service>
  <link
referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
        location="http://www.dopl.utah.gov/inspection.wsil"/>
</inspection>
```

Overall, the concept and the execution of WSIL are pretty easy to grasp. The resources box has links to WSIL references, including the specification, which contain details not appropriate for this paper.

One place where UDDI can have an impact now is within organizations since the trust relationship has already been established. UDDI allows web services to be moved from server to server without clients knowing or caring about the move in much the same way that DNS allows domains to be moved. For example, imagine that we have a payroll web service that is used by numerous other programs throughout the state. UDDI could be used to put a level of indirection between the client and the server so that the clients would not be dependent on knowing about a specific resource.

Adhere to Data Standards Such as RSS Where Available

As I mentioned before, XML is not so much a mark-up language itself as it is a framework for creating mark-up languages. As such, a number of organizations have undertaken the task of creating standard mark-up languages, based on XML for specific disciplines. As an example, the Environmental Protection Agency (EPA) has defined some XML based standards for exchanging environmental data. Also, the US Congress has defined and XML standard for bills.

Wherever possible, we should rely on such standards rather than making up our own. This ensures the greatest flexibility and reuse for our data. Organizations that you belong to may already be developing XML standards for the type of data you have. Still, dive in and keep moving; if you miss a standard it is not the end of the world because it is likely your data can be translated using XSL into whatever standards come along later.

A good example of one such general standard is Rich Site Summary or RSS. RSS produces a summary of headlines for a web resource and is especially useful for chronologically related items. Many sites such as CNN, Disney, and Slashdot provide RSS feeds. RSS feeds should be produced and advertised for chronological data such as events, press releases, rulings, judgments, decisions, and so on.

One of the nice things about RSS is that there are a number of standard programs, called RSS aggregators, which read RSS feeds and render them for human readability. As an example of how this might be useful, imagine if the events on utah.gov were also provided as RSS. Someone interested in seeing events in the State could have them appear in their RSS aggregator along with CNN headlines rather than visiting each site separately.

Another nice thing about RSS is that the State *de facto* standard for portal software, Novell Portal Product, understands RSS natively and can render it in various ways on the pages it produces. So, if the Innerweb were built as a portal and Capitol Connections, the state electronic newsletter, produced an RSS feed, the headlines from Capitol Connections could appear in a box on the Innerweb page without the Capitol Connections publisher and the portal developer having to agree on standards, have meetings about where and when the data will be available, etc. Creating the box is a simple matter and every time Capitol Connections is updated, the Innerweb page will be updated automatically.

Use HTTP Authentication as Much as Possible

Not all of the information that we want to make available as XML should be publicly available. We generally protect these kinds of resources by using authentication and authorization. Before we continue, let's define these terms:

> **Authentication** is the process of determining if you are who you say you are (i.e. are your credentials *authentic*?).
>
> **Authorization** is the process of determining, once we know who you are, whether you're allowed to have access to the resource in question.

We have made great strides, as a state, in creating a single directory tree (Utah Master Directory) that has every employee in a single namespace (utah.gov). This directory contains both authentication and authorization information. In the future, your authorization for any given resource will be determined by your job function. Your authentication will stay with you for the entire time you work for the State of Utah but your authorizations will change as you move from job to job.

Not only does this system work for state employees, but there is a directory tree for citizens and businesses so that we can provide authenticated access to Utah State web services.

We have also invested in infrastructure to allow this authentication and authorization information to be available to any web based service. So, any time you create an application that requires authentication, this infrastructure is available for you to use. Using this infrastructure is the first step in ensuring that we can offer single sign on to services on our the Innerweb and Utah.gov.

I say the first step, because even with this infrastructure, we may not be able to make use of resources if they require multiple authentication and authorization steps. One of the best ways to ensure that single sign on works across a number of resources, even before we know how those resources might be combined, is to use HTTP authentication instead of rolling your own.

HTTP authentication may not always work for an application. In particular, HTTP authentication gives the designer very little flexibility in how the authentication request is presented to the user (the browser handles it separately from a web page) and you can't cache the authentication in a cookie for later use.

With the proper design, however, it is possible to offer HTML-based authentication for people and HTTP-based authentication for machines. The key is to recognize that the URI for accessing the resource as HTML can be different from the resource for accessing the resource as XML.

A Little Computer Science Theory

There's a lot of hype surrounding XML. To read the popular press (or listen to Microsoft) you'd think it was the holy grail of computing and going to solve every problem in existence. Like most things that are heavily marketed, there's quite a bit of misunderstanding and exaggeration surrounding XML. The fact is, if you've studied Computer Science, there are some relatively simple concepts that will clear up what XML is and isn't very quickly:

- XML is a way of describing context free grammars.
- A DTD is a BNF for a particular grammar.
- XML parsers are interpreted versions of LEX and YACC.
- A DOM is a standardized parse tree.
- XSL is an interpreted pretty—printer.

These statements allow anyone with a little CS theory to separate fact from hype and make some pretty good decisions about where to deploy XML based technologies.

What about SOAP?

If you've read anything about web services, you've undoubtedly heard about SOAP. In its basic form, SOAP is an XML based remote procedure call (RPC) mechanism. With the focus in this paper on conventional web technologies, you might get the impression that I don't find SOAP useful or think we should avoid it. That is not the case. While there are certain parts of SOAP that I find problematic (like its lack of support for a universal namespace), SOAP provides a great deal that is useful, such as machine independent and language independent remote procedure calls. A future paper will discuss SOAP, its use, and how it dovetails with the principles expressed in this paper.

Conclusion

Imagine if every information or computing resource in Utah had a standard way of referencing it (URI), used a standard method of access (HTTP), included structure in its presentation (XML), enabled multiple presentation formats (XSL), had a standard way of revealing how it can be used (WRDL), advertised its presence in a standard manner (WSIL), and used single-sign on authentication and authorization compatible with every other resource (UMD). This is the real power of web services and it is easily and affordably within our reach. This vision can be realized incrementally, as we make resources available on the Internet. What's more, we can do it for little additional cost if we follow the principles and design guidance in this document.